(19)日本国特許庁（ＪＰ）　(12)**公開特許公報（Ａ）**　(11)特許出願公開番号

特開平5－197563

(43)公開日　平成５年(1993)８月６日

(51)Int.Cl.[5]	識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｇ０６Ｆ　9/45				
		9292－5B	Ｇ０６Ｆ　9/44　322　Ｆ	

審査請求　未請求　請求項の数20(全 17 頁)

(21)出願番号　特願平4－224790

(22)出願日　平成４年(1992)７月31日

(31)優先権主張番号　７４１，２９２

(32)優先日　1991年８月６日

(33)優先権主張国　米国（ＵＳ）

(71)出願人　590000400
ヒューレット・パッカード・カンパニー
アメリカ合衆国カリフォルニア州パロアルト　ハノーバー・ストリート　3000

(72)発明者　バントウォル・アール・ラウ
アメリカ合衆国カリフォルニア州ロス・アルトス・ハイランズ・サークル　900

(72)発明者　マイケル・シュランスカー
アメリカ合衆国カリフォルニア州サニーベイル・シェナンドー　1139

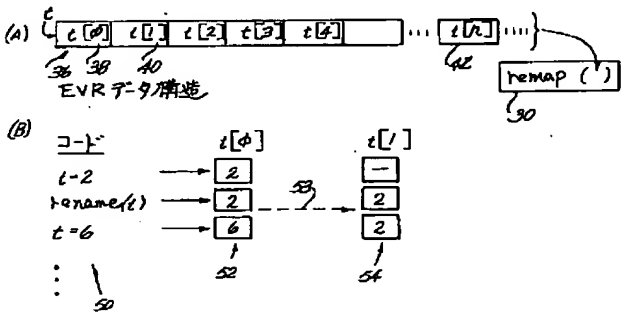
(74)代理人　弁理士　長谷川　次男

(54)【発明の名称】　最適化方法及びコンパイラ

(57)【要約】

【目的】命令レベル・パラレル・プロセッサのロード操作の最適化を行う。

【構成】静的単一代入表現（コードを静的に見た時、各仮想レジスタに１回しか代入が行われないような表現）された中間言語プログラムはループを実行すると１つの仮想レジスタに複数回の代入が行われる。そこで、当該プログラムのどの実行パスにおいても各仮想レジスタに１回しか代入が行われないような動的単一代入表現に変換する。そのため、各仮想レジスタを添字付きレジスタ配列 t[0], ..., t[n] で表すとともに、この添字と配列の実際の各要素との対応関係を１だけずらす関数 remap() を設ける。ループ中の各仮想レジスタへの代入の前に当該仮想レジスタに remap() を作用させることにより、同一の添字の仮想レジスタへの代入は１回だけになる。



1

【特許請求の範囲】

【請求項１】下記のステップ（ａ）～（ｄ）を有し、命令レベル・パラレル・プロセッサ用コンパイラの中間表現出力を最適化する方法：

（ａ）ソース・プログラムを前記コンパイラに供給する；

（ｂ）複数の拡張仮想レジスタ・データ構造を生成する：前記データ構造の各々は、個別にアクセスでき、ＥＶＲ名で識別できる複数のエレメントを有する；

（ｃ）前記ソース・プログラムを、拡張仮想レジスタ・データ構造を参照する少なくとも１つの代入分を含む単一代入中間表現に変換する；

（ｄ）下記のステップ（ｄ１）～（ｄ３）により、前記単一代入中間表現を動的単一代入中間表現に変換する：

（ｄ１）各代入文に対応して remap() 関数を出力する：remap() 関数は拡張仮想レジスタをパラメータとして受け取る；

（ｄ２）全ての remap() 関数を動的単一代入中間表現中の基本ブロックの先頭に移動する；

（ｄ３）第１の拡張仮想レジスタ・エレメントが第２の仮想拡張レジスタ・エレメントと同じメモリ中のロケーションを参照するか否かをテストし、そうであった場合には動的単一代入中間表現中の第１の拡張仮想レジスタ・エレメントを第２の拡張仮想レジスタ・エレメントで置き換えることによって動的単一代入中間表現中の冗長なロード操作を除去する。

【請求項２】前記拡張仮想レジスタ・データ構造の各々は上限なしに線型に順序付けられたメモリ中の記憶エレメントであることを特徴とする請求項１記載の最適化方法。

【請求項３】前記 remap() 関数は拡張仮想レジスタ・データ構造をパラメータとして受け入れ、前記拡張仮想レジスタの以前には第１の名前を使ってアクセスできた第１のエレメントを第２の名前を使ってアクセスできるようにすることを特徴とする請求項２記載の最適化方法。

【請求項４】前記ステップ（ｄ）は下記のステップ（ｄ４）と（ｄ５）を有することを特徴とする特許請求の範囲だい１項記載の最適化方法：

（ｄ４）複数のエイリアス集合を生成する；

（ｄ５）前記動的単一代入中間表現中の参照を、同じ配列でかつ同じ添え字についての参照ではない場合を除いて、全て１つの前記エイリアス集合中にストアする。

【請求項５】拡張仮想レジスタ・エレメントの各々は前記中間表現中の添字付き配列変数のテキスト名に結合され、

前記結合されたエレメントとテキスト名は複数のマップ・タップルの１つを構成し、

前記複数のマップ・タップルは少なくとも１つのマップ・タップル集合中に対応付けられ、

2

前記ステップ（ｄ）は、中間表現ステートメントを調べ、当該ステートメントの内容に従ってタップルを集合から消去し、タップルを集合に付加し、集合中のタップルを新たなタップルで置き換え、集合の交わりを処理するステップを含むことを特徴とする請求項４記載の最適化方法。

【請求項６】前記ステップ（ｄ）は下記のステップ（ｄ６）を含むことを特徴とする請求項５記載の最適化方法：

（ｄ６）プログラム中のある点での２つの参照が一貫してメモリ中の同一のロケーションに関連するか、決して同一のロケーションに関連しないが、一時的に関連するか、繰り返し関連するか、どの様に関連するかが不確定かを識別し、この識別結果に応答して、冗長なロードとストアを消去し、ロードとストアの順序を変更する。

【請求項７】前記ステップ（ｄ）は下記のステップ（ｄ７）～（ｄ１１）を含むことを特徴とする請求項６記載の最適化方法：

（ｄ７）正規化された形態 aJ + b の２つの添え字参照 $a_1J + b_1$, $a_2J + b_2$の入力を受け入れる：ここで、a と b はループ不変整数式であり、J はソフトウエア・ループの繰り返しの間に繰り返しその値が修正されるスカラ変数である；

（ｄ８）$a_1 - a_2 = 0$ かつ $b_1 - b_2 = 0$ か否かをテストし、そうである場合には前記２つの参照は一貫して依存関係にあるとする；

（ｄ９）$a_1 - a_2 = 0$ かつ $b_1 - b_2 \neq 0$ かあるいは $a_1 - a_2 = 0$ かつ $b_1 - b_2$ は $a_1 - a_2$ で割り切れないか、あるいは $a_1 - a_2 = 0$ かつ $b_1 - b_2$ は $a_1 - a_2$ で割り切れかつ $(a_1 - a_2)(b_1 - b_2)$ は J の値の範囲外にあるか否かをテストし（ここで # は不等号である）、もしそうであれば、前記２つの参照を独立であるとする；

（ｄ１０）ループ中で J がその値を１回しか取らないか否かをテストし、もしそうであれば前記２つの参照を過渡的に依存関係にあるとする；

（ｄ１１）$a_1 - a_2 \neq 0$、かつ $b_1 - b_2$ が $a_1 - a_2$ で割り切れ、かつ $(b_1 - b_2)/(a_1 - a_2)$ が J の値の範囲外にあり、かつ J がその値を複数回取るか否かをテストし、もしそうであれば前記２つの参照を繰り返し依存関係にあるとする。

【請求項８】プロセッサに、より少ない参照を行うようにさせ、更に参照を並列に行わせるようにするステップを含むことを特徴とする請求項７記載の最適化方法。

【請求項９】拡張仮想レジスタ・エレメントの各々は中間表現中の添字付き配列変数のテキスト名に結合され、前記結合されたエレメントとテキスト名は複数のマップ・タップルの１つを構成することを特徴とする請求項１記載の最適化方法。

【請求項１０】前記マップ・タップルは少なくとも１つ

のマップ・タップル集合の要素であることを特徴とする請求項９記載の最適化方法。

【請求項１１】下記の（ａ）ないし（ｃ）を設けてなる命令レベル・パラレル・プロセッサ用コンパイラ：

（ａ）メモリ中に複数の拡張仮想レジスタ・データ構造を生成する手段：前記拡張仮想レジスタ・データ構造の各々は線形に順序付けされた複数の仮想レジスタ・エレメントを有する；

（ｂ）テキスト名と実メモリ・ロケーションをメモリ中の前記拡張仮想レジスタ・エレメントに対応つける手段；

（ｃ）複数のテキスト名を前記拡張仮想レジスタ・エレメントの１つに対応つける手段。

【請求項１２】前記拡張仮想レジスタ・データ構造をスカラ変数としてアクセスする手段を設け、前記拡張仮想レジスタ・エレメントが個別的にアクセスできることを特徴とする請求項１１記載のコンパイラ。

【請求項１３】前記拡張仮想レジスタ・データ構造は関数 remap() に応答し、前記関数 remap() は以前には前記拡張仮想レジスタ・エレメントのうちの第１の名前を使ってアクセスできたものを第２の名前を使ってアクセスできるようにすることを特徴とする請求項１２記載のコンパイラ。

【請求項１４】下記のステップ（ａ）を有する命令レベル・パラレル・プロセッサ用コンパイラの中間表現出力を最適化する方法：

（ａ）ソース・プログラムの中間表現を動的単一代入中間表現に変換する：前記動的単一代入中間表現では与えられた前記中間表現中の複数の代入ステートメントが拡張仮想レジスタ・データ構造を参照する複数の縮約化されたステートメントに変換されている。

【請求項１５】前記ステップ（ａ）は、動的単一代入形態への変換のステップ中に、前記中間表現からコピー操作を削除するサブステップを有することを特徴とする請求項１４記載の最適化方法。

【請求項１６】重複したレジスタ代入参照を拡張仮想レジスタ中の複数のレジスタ・エレメントのうちの１つへの参照で置換するステップを有することを特徴とする請求項１５記載の最適化方法。

【請求項１７】以下のサブステップ（ｂ）～（ｆ）を有することを特徴とする請求項１６記載の最適化方法：

（ｂ）標準化形態 aJ + b で表したとき $a_1J + b_1$, $a_2J + b_2$ となる２つの入力添字参照を受け付ける：ここで a と b はループ不変整数式であり、J はソフトウエア・ループの繰り返し中にその値が繰り返し変更されるスカラ変数である；

（ｃ）$(a_1 - a_2 = 0)$ かつ $(b_1 - b_2 = 0)$ が成立するか否かを試験し、これが成立した場合は、これらの参照を「一貫して依存関係にある」とする；

（ｄ）$((a_1 - a_2 = 0)$ かつ $(b_1 - b_2 \neq 0))$ または $((a_1 - a_2 \neq 0)$ かつ $((b_1 - b_2)$ が $(a_1 - a_2)$ で割り切れる）または $((a_1 - a_2 \neq 0)$ かつ $((b_1 - b_2)$ が $(a_1 - a_2)$ で割り切れない）かつ $((b_1 - b_2)(a_1 - a_2)$ が J の値の範囲外にある）が成立するか否かを試験し、これが成立した場合は、これらの参照を「独立である」とする；

（ｅ）J がループ中にその値を１回だけ取るか否かを試験し、それが成立した場合はこれらの参照を「過渡的に依存関係にある」とする；

（ｆ）$((a_1 - a_2 \neq 0)$ かつ $((b_1 - b_2)$ が $(a_1 - a_2)$ で割り切れる）かつ $((b_1 - b_2)(a_1 - a_2)$ が J の値の範囲内にある）かつ（J がその値を複数回取る）が成立するか否かを試験し、それが成立した場合は「繰り返し依存関係にある」とする。

【請求項１８】プロセッサがより少ない回数しか参照を行わないようにし、かつ参照を並列に行うようにするステップを含むことを特徴とする請求項１７記載の最適化方法。

【請求項１９】拡張仮想レジスタ・エレメントの各々は前記中間表現中の添字付き配列変数のテキスト名と組み合わされ、前記組み合わされたレジスタ・エレメントとテキスト名は複数のマップ・タップルの１つを構成することを特徴とする請求項１８記載の最適化方法。

【請求項２０】前記レジスタ・エレメントとテキスト名は複数のマップ・タップルの１つを構成し、複数のマップ・タップルは少なくとも１つのマップ・タップル集合の要素であることを特徴とする請求項１９記載の最適化方法。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】本発明は、一般に電子計算機用のプログラミング言語コンパイラに関するものである。特に本発明は、命令レベル・パラレル・プロセッサ用コンパイラにおいて入力ソース・コードが添字付き配列変数への複数参照を含む場合にコンパイラが生成するオブジェクト・コードを最適化するための方法に関するものである。

【０００２】

【従来技術及びその問題点】最初の頃のコンピュータは、プロセッサに対して１つの命令ストリームと１つのデータ・ストリームを設けるいわゆるフォン・ノイマン・プロセッサ・アーキテクチャを使用した。このプロセッサは、各命令を順次実行し、その際１つの記憶領域のデータに対して処理動作を行う。このような古典的プロセッサは、単一命令単一データ（ＳＩＳＤ）プロセッサとして知られている。プロセッサのクロック速度が速くなるに従って、ＳＩＳＤアーキテクチャは高い処理スループットを達成することの障害となってきた。多くのコンピュータプログラムは、個別ではなく並列に実行することができる命令のグループまたはブロックを有する。

そのため、コンピュータの研究者達は、並列処理モデルのファミリーを含む他のアーキテクチャを開発するに至った。

【０００３】Ａ．命令レベル・パラレル・プロセッサ
パラレル・プロセッサの１つのタイプは命令レベル・パラレル（InstructionLevel Parallel、ＩＬＰ）プロセッサとして知られている。ＩＬＰプロセッサにおいては、スケジューリングや同期化を行う計算の基本単位は、個々の加算（add）、乗算（multiply）、ロード（load）またはストア（store）操作のようなプロセッサ命令である。相互依存しない命令は並列にロードされ実行される。

【０００４】ＩＬＰプロセッサを用いると、命令スケジューリングあるいは同期化はプログラム実行の間に行う必要がなく、一部の決定はプログラムのコンパイル時に行うことができる。例えば、コンパイラが２つの操作が独立している（どちらも他方の操作の結果を入力として必要としない）ことを確認できるのであれば、それらの操作は並列に実行することができる。ＩＬＰプロセッサの動作を最適化する１つの方法は、実行時決定は全く不可能であるという仮定の下にコードを生成するコンパイラを作ることである。このコンパイラは、全てのスケジューリング及び同期化をおこなう。従って、プロセッサは実行時にはコードの順序変更（re-ordering）をほとんど行わず、これによってリンク及び実行速度が改善される。

【０００５】Ｂ．データフロー・プロセッサ
もう１つのパラレル・プロセッサのタイプはデータフロープロセッサであり、このタイプは最大限に並列化されたプログラム実行が可能である。データフロー・プロセッサにおいては、命令は、必要データが全て処理に使える状態になるとすぐ実行される。データフロー・アーキテクチャ及びそのためのコンパイラは以下の参考文献に説明されている：

J. B. Dennisの"First Version of a Data Flow Procedure Language", Proceedings of Programming Sumposium, Paris, 1974 (Lecture Notes in ComputerScience 19(Springer-Verlag、Berlin, 1974d)に再録)

K. R. Traub の"A compiler for the MIT Tagged-Token Dataflow Architecture", Technical Report MIT/LCS/TR-370 (MIT Laboratory for Computer Science, Cambridge, Mass., August, 1986)

【０００６】Ｃ．ベクトル・プロセッサ
ベクトル・プロセッサも特定の形式の並列プログラムを実行することができる。当技術分野においては周知のように、ベクトル命令は、ループ本体中に単一の操作を有するループと等価である。例えば、ベクトル V のベクトル累算命令：

```
V := ++1;
```

は、以下の疑似コード・ループと等価である：

```
for Index := 1 to ベクトル長 do V[Index] := ++1
```

その結果、ベクトル・プロセッサ用コンパイラにおけるオブジェクト・コード生成は、主としてループ構造をベクトル命令に変換することに力を入れる。このようなベクトルコードの最適化は配列の添字の詳細な解析を必要とする。例えば、コンパイラは、２つの参照が同じ実メモリ・ロケーションに対するものであるかどうかを判断し、それらの参照を統合しなければならない。ベクトル・コンパイラの従来技術に関するまとめは、H. Zima 及び B. Chapman の"Supercompilersfor Parallel and Vector Computers" (ACM Press, Association for Computing Machinery, New York, 1990) に記載されている。

【０００７】残念なことには、これらの従来技術のベクトル・プロセッサ用コンパイラは、（冗長なあるいは部分的に冗長な配列参照の除去のような）最適化より、むしろ（ループ交換や分配（loop interchange and distribution）のような）プログラム変換に焦点が絞られていた。例えば、コード・ブロック１ＡのＦＯＲＴＲＡＮコード・ループは B(I-1) の冗長なロードを行っている。

○コード・ブロック１Ａ

```
DO 10 I = 2, 50
A(I) = B(I) + B(I-1)
10 CONTINUE
```

ベクトル・コンパイラは、ベクトル B の要素の２つのロードが１インデクス・エレメントだけスキューされているということについては、典型的なベクトル・プロセッサのハードウェアでは利用することができないため、これを認識しようとはしない。しかしながら、当技術分野においては周知のように、コード・ブロック１Ｂの形式のコードを用いて冗長なロードが除去されるならば、ＩＬＰプロセッサはより効率的となる。

○コード・ブロック１Ｂ

```
T = B(1)
DO 10 I = 2, 50
S = B(I)
A(I) = S + T
T = S
10 CONTINUE
```

【０００８】コード・ブロック１Ｂを使用した場合には、ＩＬＰプロセッサは４番目と５番目のステートメントを同時に実行することができ、これによって１回のループ反復につきロード操作が１つ少なくなるので、性能が改善される。このように、ベクトル・プロセッサ最適化技術はＩＬＰプロセッサには適しない。

【０００９】Ｄ．従来技術の欠点
従来のＳＩＳＤプロセッサでは、処理または問題解決における並行性の機会がほとんど得られない。従来の大部分のコンパイラはＳＩＳＤプロセッサ用に書かれているので、パラレル・プロセッサのための最適化はなされて

いない。一方、従来の大部分のコンパイラは、任意の制御フロー（すなわちＧＯＴＯ型の分岐命令及び構造化されていない繰り返し制御フロー）を持つプログラムを解析し、最適化することができる。

【００１０】ＩＬＰプロセッサ及びコンパイラは、分岐命令が決定するところに従い、概ね逐次的に、１つの基本オブジェクト・コード・ブロックからべつの基本オブジェクト・コード・ブロックへ進む制御フローを与える。しかしながら、ＩＬＰコンパイラは、一部の命令を並列に動作させることによってスカラ性能を改善する。例えば、ＩＬＰコンパイラは、最も内側のループ・レベル内の有限個の連続した命令を並列実行させることによって、ループ構造を最適化することができる。この最適化のためには、コンパイラは、並列に実行される程度に接近した一連の命令の間の相互依存関係を解析するだけでよい。特に、２つの命令間の依存関係の性質は、この依存関係が侵犯される、つまり１つの演算からもう一方の演算へ結果がレジスタで持ち込まれる場合においてのみ重要である。

【００１１】例えば、依存関係解析は、コード・ブロック２Ａのように同じネスティング・レベルの複数の「兄弟関係」ループを有するコード・ブロックの場合と、コード・ブロック２Ｂに示すような依存関係を持つロードが入っているループでは異なる。

```
○コード・ブロック２Ａ
DO 10 I = 1, N
X(I+K) = ...  [ストア]
10 CONTINUE
.
.
.
DO 20 I = 1, N
... = X(I+L)    [ロード]
20   CONTINUE
○コード・ブロック２Ｂ
DO 30 I = 1, N
10   ... = X(I-K)  [ロード]
.
.
.
20  X(I) =  ...    [ストア]
30  CONTINUE
```

コード・ブロック２Ａのプログラム・セグメントをコンパイルしているときは、ステートメント１０と２０の２つのループのストア・ステートメントとロード・ステートメントの間の依存関係は全く問題とならない。これら２つのループは基本ブロックであるため、ＩＬＰコンパイラはこれらのループの順序を守るので、これら２つのループが逆の順序で実行されるようスケジューリングする危険性はない。また、ベクトル X が小さいということが既知でない限り、ＣＰＵのレジスタは、第１のループにおけるロード操作によってそこにストアされた値を全て保持して第２のループにおけるロードを除去するには容量が通常不充分であるため、最適化はほぼ不可能である。

【００１２】同様に、コード・ブロック２Ｂのロードとストアをコンパイルするとき、ロード操作とストア操作がコード中ではるかに離れている場合は、これらの操作が通常依存関係にあるかどうかはどうでもよい。操作間の距離がこのように大きいと、これらの操作が逆順にスケジューリングされることは全くない。従って、最大の関心事は、これらの操作が時間的に互いに近接しているかどうかである。

【００１３】しかしながら、部分的な依存情報でも、ＩＬＰコンパイラの性能に役立つ。多数の反復によって隔離されている操作については証明が得られない場合でも、コンパイラが隣合った繰り返し中の操作が互いに独立していることを証明し得るならば、性能を著しく向上させることができる。このことは、ベクトル化コンパイラあるいは並列化コンパイラについては当てはまらない。

【００１４】例えば、コード・ブロック２Ｂにおいて、ステートメント２０のストアがステートメント１０のロードに依存するものと仮定する。スカラ変数 K がゼロか負であると、繰返しは不可能である。K がゼロより大きいならば、DO 30 のループの与所の繰り返しにおける行１０のロードは、繰り返しの回数が K 回前の行２０のストアに依存する。K が未知の場合は、コンパイラは最悪のケース（K = 1)を想定しなければならず、隣り合った繰り返しのオーバラップは全く不可能である。コンパイラが K > 1 であるということだけでも証明できれば、この２倍の性能を達成することができる。控え目に仮定して、K = 2 であり、隣り合った繰り返しの開始頻度をほぼ２倍にすることができる。

【００１５】残念なことには、ＩＬＰアーキテクチャは従来の高水準コンピュータプログラミング言語コンパイラ及びコンパイラ最適化技法に充分適してはいない。当技術分野においては周知のように、ソース言語コンパイラは、最適化及びオブジェクト・コード生成前に中間表現（Intermediate Representation、ＩＲ）として知られている中間コードを作る。しかしながら、大部分の従来技術のＩＲは命令レベル並行処理を表現することができない。更に、従来技術のコンパイラは添字付き変数に対する参照の良好な最適化を行っていない。

【００１６】

【目的】従来のコンパイラ及びオプティマイザ（optimizer）では、ＩＬＰプロセッサにとって最大限効率的なコードが生成されない。従って、プロセッサ及びコンパイラの設計者は、添字付きの配列変数に対する参照を処理するための最大限に効率的なＩＬＰプロセッサコード

を生成するＩＬＰコンパイラ最適化方法があれば有用であろうということを理解できるだろう。

【００１７】また、このような設計者は、従来のスカラ変数技法の力を配列への添字付き参照を含むあるクラスのプログラムに適用した最適化方法の有用性も理解できるであろう。

【００１８】また、従来のコンパイラにより生成される中間表現はＩＬＰアーキテクチャに余り適してはいない。従来のオプティマイザは、構造化ループについては効果的であるが、条件付き分岐命令が埋め込まれているループのような構造化されていない繰り返し構造に適用すると劣悪なコードを生じる。従って、プロセッサやコンパイラの設計者は、任意の繰り返し制御フローを持つ最も内側のループで使用する最適化方法が有用であることを理解できよう。

【００１９】また、プロセッサ及びコンパイラの設計者は、命令レベル並行処理のために最適化した依存関係解析を行うことが可能なコンパイラが得られるならば、これも評価するであろう。

【００２０】データフロー・モデルはＩＬＰコンパイラに好適なＩＲ生成のための基礎を与えてくれる。しかしながら、データフローＩＲは、従来の高水準言語プログラム中で与えられる任意の構造化されていない制御フロー・パスを扱うように拡張しなければならない。更に、データフロー・モデルは、並列的にデータを使用できるようにするために過大な数のコピー操作を必要とする。当業者であれば、最適化法を用いて不必要なコピー操作を除去するＩＬＰ最適化方法が有用であろうということを理解できるであろう。

【００２１】また、コンパイラやプロセッサの設計者は、ループを含んだ任意のフローグラフの存在下で添字付きのメモリ・ロケーション間の関係を含む操作間の依存関係を最適化し、また不必要なコピー操作のない最適化方法が有用であることを理解するであろう。また、主要な注意が添字付き参照に対して確実に払われるようにすることも好都合であろう。

【００２２】

【概要】従って、本発明は、ＩＬＰプロセッサ用のコンパイラ中間表現（ＩＲ）を最適化するためのプロセス、及びこのプロセスを実現するためのデータ構造を提供するものである。本発明のプロセスは、ランダム・アクセス・メモリのようなメモリ記憶手段へのアクセス及び磁気ディスク記憶装置のようなプログラム用大容量記憶手段へのアクセスを有する電子計算機またはデータ処理装置上で動作するコンパイラ・コンピュータ・プログラムとして実施することが望ましい。このコンパイラ・プログラムは、プログラム用大容量記憶手段に記憶された入力ソース・プログラムを読み込み、疑似機械命令を用いてメモリ中にソース・プログラムの動的単一代入中間表現（dynamic single assignment intermediate representation）を生成する。動的単一代入中間表現を生成するために、コンパイルの間に、コンパイラは、ソース・プログラム中で定義されている変数を記憶するためにメモリ中に複数の仮想レジスタを生成する。このプロセスにより、コンパイラは、ＩＲコードを文字通り読んだ場合には左辺に同じ仮想レジスタがある代入ステートメントが複数本ある場合であっても、同じ仮想レジスタに対しどの動的実行パス上でも決して複数回の代入が行われることがないようにする。このプロセスは、無限の、直線状に配列された仮想レジスタ・エレメントの集合よりなるＥＶＲデータ構造を用いて実現され、拡張仮想レジスタ（ＥＶＲ）上で remap() 関数が定義されている。ＥＶＲパラメータを付けて remap() 関数を呼び出すと、リマップ操作前には [n] としてアクセス可能であったＥＶＲエレメントがリマップ操作後は [n+1] としてアクセス可能となる。このプロセスにより、プロセッサは、動的な複数のマップ・タップル（map tuple）よりなる添字付き参照マップを生成して使用する。各マップ・タップルは、テキスト表現された名前（textual name）でアクセス可能な実メモリ・ロケーションをＥＶＲエレメントと対応付ける。このようにして、コンパイラは、マップ・タップルを用いてテキスト表現された名前をＥＶＲエレメントで置換することができ、これによって出力の中間表現から不必要なロード操作を除去することができる。

【００２３】

【実施例】以下の本発明の実施例の詳細な説明においては、説明をわかり易くするために一部特殊な用語を使用する。しかしながら、本発明は、ここで使用する特定の用語に限定されるものではなく、実質的に同様の結果を達成するために実質的に同様な動作をする技術的に均等な事柄全て含むものとする。本願の技術分野の当業者であれば、コンパイラの理論と実践に関する教科書で論じられている全ての概念及び技術を理解できるであろう。そのような教科書の１つに、Addision-Wesley (Reading, Mass.) によって１９８５年に出版された A. Aho, R. Sethi と J.Ullman 共著の "Compilers: Principles, Techniques and Tools" があり、その用語及び定義は本願明細書でも使用する。

【００２４】本発明は、コンパイラの中間表現（ＩＲ）コードを最適化するための方法を提供すると共に、その方法を実施するためのデータ構造を提供するものである。好ましくは、図１に示すように、本発明の方法は、ランダム・アクセス・メモリのようなメモリ・ストレージ手段（メモリ）４へ、また磁気ディスク・ストレージのようなプログラム用大容量ストレージ手段（ディスク・ストレージ）６へアクセスできる電子計算機またはデータ処理装置（プロセッサ）２上で動作する従来のコンパイラ・コンピュータ・プログラム８を用いて実施する。まず始めに、コンパイラ８は、プログラム用大容量

ストレージ手段に記憶されたＦＯＲＴＲＡＮプログラミング言語で書かれたプログラムのような入力ソース・プログラム１０を読み込み、以下に説明する方法により疑似機械命令を用いてメモリ中にソース・プログラムの動的単一代入中間表現（Dynamic Single Assignment Intermediate Representation, ＤＳＡ ＩＲ）１２を生成する。

【００２５】Ａ．依存関係解析及び単一代入

ＩＬＰコンパイラが最大限の並列プログラム表現を達成するためには、正しい意味を確保するのに本質的ではない命令依存性を全て除去することが必要である。当技術分野においては周知であり、また H. Zima と B. Chapman 共著の "Supercompilers for Parallel and Vector Computer" (ACM Press, New York) に開示されているように、命令相互依存関係（instruction inter-dependency）は、真の依存関係（true dependency）、反依存関係（anti-dependency）及び出力依存関係（output dependency）に分類することができる。正しい意味のためには真の依存関係のみが本質的である。反依存関係及び出力依存関係は、いくつもの操作の結果を同じ変数に対して代入する結果生じるものであり、本発明を使用する場合はこのような結果が現れることは決してない。

【００２６】並列コンパイラの中間表現（ＩＲ）においては、反依存関係及び出力依存関係は、コンパイラが結果を仮想レジスタに決して２回以上代入することがないようにすることにより除去することができる。この「単一代入（single assignment)」の原理は、L. Tesler と H. Enea 共著の "A Language Design for Concurrent Processes", Proceedings AFIPS Sprint Joint Computer Conference (1968) ４０３～４０８ページに開示されている。単一代入は、上に引用した A. Aho 他の教科書に開示されている値番号付け手続（value numbering procedure）を用いて単一の基本コード・ブロックで実現することができる。

【００２７】当技術分野においては周知のように、コンパイラは、代入操作がＩＲに入れられる毎にメモリ４に別個の仮想レジスタ１４を生成するつまり設定することによって、ソース・プログラムの静的単一代入ＩＲを得ることができる。静的単一代入ＩＲにおいては、各仮想レジスタはプログラム中で代入の左辺にちょうど１回だけ現れる。しかしながら、このような代入がループ構造中に起こった場合には、ループを何回も反復すると、同じ仮想レジスタに対して何回も代入が行われる。そのような複数回の代入は動的単一代入の原理に違反し、ループ中の操作間に反依存関係を生じさせる。更に、特定の仮想レジスタの多重定義を区別することは不可能であり、そのために依存関係を精密に指定することは不可能になる。従って、当業者であれば、動的な意味で単一代入を可能にすることによってこれらの問題を解消するＩＲを高く評価するであろう。

【００２８】Ｂ．動的単一代入

並行性を強化し、ループ構造内で静的単一代入を用いることの問題を解消するための１つの方法は、ＩＲを動的単一代入（ＤＳＡ）形式にすることによるものである。

【００２９】図３に示すように、コンパイラ８でのＩＲ最適化には、ブロック６０で始まるプロセスを使用する。本発明の１つの特徴は、本願中において動的単一代入（ＤＳＡ）と定義し、図４に詳細に示すＩＲ最適化方法のステップ６２である。ＩＲをＤＳＡ法によって作成すると、文字通りに読んだ場合には、ＩＲコード中に左辺に同じ仮想レジスタがある複数の代入ステートメントがある場合でも、同じ仮想レジスタにはどの動的実行パスについても決して２回以上代入されることはない。この情況は静的単一代入とちょうど正反対である。静的単一代入では、ひとつの仮想レジスタはステートメントの左辺に２回以上現れることは決してないが、ループ内ではその仮想レジスタに対して必然的に何回も代入が行われる。

【００３０】本発明においては、入力ソース・プログラム１０は、３つの基礎的なツール、すなわち拡張仮想レジスタ（ＥＶＲ）データ構造、remap() 関数、及び複数のマップ・タップルよりなるメモリ・マップ、を用いて動的単一代入ＩＲ１２に変換される。これらのツールは、図３のステップ６６、６７及び６８を実行するために用いられる。

【００３１】図２（Ａ）に示すように、ＥＶＲデータ構造３６は、好ましくは、無限の直線状に配列された仮想レジスタ・エレメント３８、４０、４２の集合よりなり、特別の操作３０、すなわち remap() がその上で定義されている。remap() 関数の唯一のパラメータはＥＶＲの名前である。ＥＶＲに t という名前が付いていれば、ＥＶＲのエレメントは t[n] でアドレス指定し、読み出し、書き込むことができる。ただし、ｎは任意の整数である。ここでは、便宜上最初のエレメント t[0] が t とも呼ばれる。remap(t) 関数を呼び出すと、remap(t) 操作の前にt[n] としてアクセス可能なエレメントがあった場合、そのエレメントは remap(t)操作の後では t[n+1] としてアクセス可能となる。ここで、ラベル t は下記のコード・ブロック３Ｂ～３Ｆの仮想レジスタ名に対応付けるために意図的に用いたものである。

【００３２】実際、remap() 関数は、拡張仮想レジスタの１つのエレメントを別のエレメントのＥＶＲ名を用いてアクセスできるようにする。ＥＶＲ t への結果の連続した代入の間に remap(t) が実行されると、t に代入された前回の値は t[1] に入っており、今度実行されるときの代入は別のエレメント t[0] に対して行われる。図２（Ｂ）は、この操作を、ＩＲコード・セグメント５０と、参照番号５２が付けられたエレメント t[0] の内容と、参照番号５４のエレメント t[1] の内容を並置することで示している。オプティマイザが命令 [t=2] に

13

遭遇すると、値[2] をエレメント t[0] にストアする式がオプティマイザＩＲ出力に付加される。次に、remap (t) 関数がＩＲ出力に付加される。remap() 関数は、矢印５３で示すように、 t[０] の内容を t[1] へ移動させる。次に、命令 [t=6] によって値 [6] が t[0] にストアされる。プログラム・テキスト上は、結果が t[0] に繰り返し代入されているように見えるが、その時々に異なるＥＶＲエレメント及び実メモリ・ロケーションへストアすることによって、動的に単一代入ルールが守られている。

【００３３】remap() 関数は、図３のブロック６４に示すように、ソース・プログラムの最適化及びＩＲへの変換の間にＤＳＡ ＩＲ１２に付加される。ＩＲ中での各代入の前に１つの remap() ステートメントが付加される。

【００３４】ＥＶＲ及び remap() 関数を使用して、ソフトウェア・ループで動的単一代入を達成するためのプログラムがコード・ブロック３Ａ～３Ｆ及び図４に示されている。

【００３５】コード・ブロック３Ａは、ＦＯＲＴＲＡＮ言語による簡単なループを示す。例として、コード・ブロック３Ａは、本発明を組み込んだコンパイラ８に入力されるソース・プログラム１０の一部を形成することができる。

【００３６】コード・ブロック３Ｂは、ステップ７２で作成されるコード・ブロック３Ａのループに対応する逐次的ＩＲである。

【００３７】コード・ブロック３Ｃは、コード・ブロック３Ａ及びステップ７４の単一代入ＩＲである。

【００３８】コード・ブロック３Ｄは、ステップ７６に示すように、ループ本体内の各代入の前に remap() 操作を挿入することによって得られる、コード・ブロック３Ｂの逐次的ＩＲについての正準動的単一代入ＩＲである。

【００３９】コード・ブロック３Ｅは、全ての remap() 操作をループの始めへ移動させた（ステップ７８）後の、ＤＳＡ形式のコード・ブロック３Ｄを示している。

【００４０】コード・ブロック３Ｆは、コピー最適化を実行した（ステップ７９）後のコード・ブロック３Ｅを示す。

【００４１】

14

コード・ブロック３Ａ（ＦＯＲＴＲＡＮループ）

```
K = 0
J = 0
DO 10 I=1, 50
    L = J
    J = J + K
    K = L
10 CONTINUE

    K = J + 5
```

コード・ブロック３Ｂ（逐次的ＩＲ）

```
                % t00 = 0, t01 = 1
                % t02 = 5, t03 = 50
s0  t04 = copy(t00)
s1  t05 = copy(t01)
s2  t06 = copy(t01)
s3  t07 = copy(t05)
s4  t05 = iadd(t05, t04)
s5  t04 = copy(t07)
s6  t06 = iadd(t06, t01)
s7  t08 = ile(t06, t03)
s8      brt(t08, s03)
s9  t09 = iadd(t5, t2)
```

コード・ブロック３Ｃ（静的単一代入ＩＲ）

```
                % t00 = 0, t01 = 1
                % t02 = 5, t03 = 50
K = 0           s0  t04 = copy(t00)
J = 1           s1  t05 = copy(t01)
DO 10 i=1, 50   s2  t06 = copy(t01)
                s10 t10 = 0 (t04, t14)
                s11 t11 = 0 (t05, t13)
                s12 t12 = 0 (t06, t15)
    L = J       s3  t07 = copy(t11)
    J = J + K   s4  t13 = iadd(t11, t10)
    K = L       s5  t14 = copy(t07)
10 CONTINUE     s6  t15 = iadd(t12, t01)
                s7  t08 = ile(t15, t03)
                s8      brt(t08, s10)
    K = J + 5   s9  t9 = iadd(t13, t02)
```

15

コード・ブロック３Ｄ
（正準動的単一代入）

```
                    % t0 = 0, t1 = 1
                    % t2 = 5, t3 = 50
K = 0               s0  t4 = copy(t0)
J = 0               s1  t5 = copy(t1)
DO 10 I=1, 50       s2  t6 = copy(t1)
                    s10 remap(t7)
    L = J           s3  t7 = copy(t5)
                    s11 remap(t5)
    J = J + K       s4  t5 = iadd(t5[1], t4)
                    s12 remap(t4)
    K = L           s5  t4 = copy(t7)
                    s13 remap(t6)
10 CONTINUE         s6  t6 = iadd(t6[1], t1)
                    s14 remap(t8)
                    s7  t8 = ile(t6, t3)
                    s8  brt(t8, s10)
    K = J + 5       s9  t9 = iadd(t5, t2)
```

コード・ブロック３Ｅ
（remap()の移動後）

```
                    % t0 = 0, t1 = 1
                    % t2 = 5, t3 = 50
K = 0               s0  t4 = copy(t0)
J = 0               s1  t5 = copy(t1)
DO 10 I=1, 50       s2  t6 = copy(t1)
                    s10 remap(t7)
                    s11 remap(t5)
                    s12 remap(t4)
                    s13 remap(t6)
                    s14 remap(t8)
    L = J           s3  t7 = copy(t5[1])
    J = J + K       s4  t5 = iadd(t5[1], t4[1])
    K = L           s5  t4 = copy(t7)
10 CONTINUE         s6  t6 = iadd(t6[1], t1)
                    s7  t8 = ile(t6, t3)
                    s8  brt (t8, s10)
    K = J + 5       s9  t9 = iadd(t5, t2)
```

16

コード・ブロック３Ｆ
（コピー最適化後）

```
                    % t0 = 0, t1 = 1
                    % t2 = 5, t3 = 50
                s0  t4 = copy(t0)
                s1  t5 = copy(t1)
                s2  t6 = copy(t1)
                s11 remap(t5)
                s13 remap(t6)
                s14 remap(t8)
                s4  t5 = iadd(t5[1], t5[2])
                s6  t6 = iadd(t6[1], t[1])
                s7  t8 = ile(t6, t3)
                s8  brt (t8, s10)
                s9  iadd(t5, t2)
```

【００４２】最初の５行の初期化ステップを除いては、コード・ブロック３Ｂ～３Ｆの各行は、コード・ブロック３Ａのソース・コードの行と対応する。全ての操作のソース及びデスティネーションは、メモリ４にストアされ、t1 ～ t10 のラベルが付与された仮想レジスタ１４である。当技術分野においては周知のように、上記コード・ブロックの "iadd" 命令は、ＣＰＵの整数加算操作を表し、"ile" 命令は整数型の以下か否かの比較であり、"brt" 命令は、「真ならば分岐せよ (branch if true)」の操作である。コード・ブロック３Ａのループにより、ＩＲは、番号をつけられたステートメントへの分岐参照ができるようにするためにステートメント番号 s0 ～ s14 を用いる必要がある。

【００４３】コード・ブロック３Ｂの逐次的中間表現はステップ７２で生成され、同じ仮想レジスタへの代入の場合がいくつか含まれている。例えば、仮想レジスタ t04と t06 に対してはブロック３Ａ中では代入が２度行われる。レジスタ t4 はステートメント s0 及び s5 で代入があり、レジスタ t5 はステートメント s1 及び s4 で代入がなされ、レジスタ t6 はステートメント s2 及び s6 で参照される。前に述べたように、これは反依存関係を持ち込み、並行性を妨げるので好ましくない。

【００４４】同じ仮想レジスタへの反復代入は、ステップ７４で作成されるコード・ブロック３Ｃの静的単一代入ＩＲを用いて排除することができる。しかしながら、より多くの仮想レジスタが必要であり、コード・ブロック３Ｂと３Ｃには同数のロード操作がある。

【００４５】反依存関係及び反復代入は、どちらもステップ７６で生成されるコード・ブロック３Ｄの正準ＤＳＡ形式を用いて取り除くことができる。この形式への変換は、s0 から s2 までのステートメント の左辺の t4、t5、t6 への参照を t4[1]、t5[1]、t6[1] への参照で置換することにより行われる。コード・ブロック３ＤのＩＲには、仮想レジスタへの各代入の前に remap() 操作が挿入される。ステートメント s4 で t5 を使用す

る前に t5 に対する remap() 操作を挿入するので、s4 の右辺の t5 への参照を t5[1] に変えることが必要となる。同様に、ステートメント s6 は t6[1] を使うように変更される。しかしながら、コード・ブロック３Ｄは、t4、t5 及び t6 が各々ステートメントの左辺に２度ずつ現れるので、まだ静的単一代入ルールに違反している。

【００４６】従って、ステップ７８に示すように、好ましくは、全ての remap() 操作をループ本体の先頭に移動させることにより更に形式的変換を行い、コード・ブロック３ＥのＩＲを得る。t4 及び t5 に対する remap() をステートメント s4 及びs3 の前に動かすことにより、右辺の参照は t4[1] 及び t5[1] に変わる。

【００４７】次に、意味のないコピー操作を取り除くことによって、当技術分野において周知の方法によりコピー最適化をステップ７９において行うことができる。コード・ブロック３Ｅは、t7 が常に t5[1] と同じ値を有し、t4[1] が常に t7[1] と同じ値を有し、他方 t7[1] は t5[2] と同じ値を有するということに着目することによって、コピー最適化されたコード・ブロック３Ｆに変換することができる。t7 及び t4 を各々 t5[1] 及び t5[2] で置換した後は、コード・ブロック３Ｅのステートメント s3 及び s5 はデッド・コードであり、コード・ブロック３Ｆにおいては省略してもよい。その結果、ステートメント s10 及び s12 は不必要である。コード・ブロック３Ｅのステートメント s4 には、今回の繰り返しに関する加算の結果が前の２回の繰り返しに関する同じステートメントの結果の和であるということが実効的に記述されている。この表記法によれば、コード・ブロック３Ｂをより短く、より効率的な方法で表すことが可能である。

【００４８】Ｃ．添字付き配列変数（subscripted array variable）の解析

本発明によれば、添字付きの配列変数の参照は、コンパイラにスカラ変数参照の場合と全く同様に配列参照を扱わせることによって最適化することができる。当技術分野においては周知のように、X(I,J) のような配列の要素は、同じ配列要素のメモリ・ロケーションを、例えば X(I,J), X(I+1,J-1) あるいは X(L,M) 等のように種々のやり方で参照することができるという点を除けば、スカラ変数と何ら異ならない。I, J, L, M の値が適切であれば、これら３通りの参照はいずれも同じメモリ・ロケーションを指示する。

【００４９】図４のステップを使って図３のステップ６２でＤＳＡ形式を確立した後、コンパイラの配列参照の最適化は、図３のステップ６７に示すような依存性解析によって行うことができる。依存性解析は、図６及び図７のデータフロー解析ステップを用いて行われる。重要なステップは、図６でテストされるように、コンパイラに２つの参照が同じメモリ・ロケーションを指しているということを認識させることである。プログラム中の、I = I+1 の形式の累算ステートメントの直前のある点に X(I) があるということをコンパイラが知れば、X(I-1) はその累算ステートメントの後では、累算ステートメントより前の X(I) と同じメモリ・ロケーションを指す。

【００５０】好ましくは、同じ配列に対する全ての参照の添字は、aJ+b の形式となるように、そこまでのコンパイラ・ステップで正規化されているものと仮定する。ここで、a, b はループ不変式であり、J はループ内で繰り返し修正される何らかの整数値を取るスカラ変数である。aJ+b の形式の添字では、J は添字変数と呼ばれる（J は一般にはループ・インデクスである）。同じ配列に対する全ての参照の添字は同じ変数 J を用いて表すことが好ましい。

【００５１】ループ内では、J のような添字変数の定義は全て $J = J+K_i$ の形式とすることが好ましい。ここで、これらの K_i はループ不変整数量である。ループ内にはこのような添字変数の定義が複数あってもよい。

【００５２】添字付き参照の依存性解析

本発明の配列最適化方法には、ブロック６７（図３）及び図５に示すように、添字付き参照の依存性解析を行うことが含まれる。

【００５３】まず、別名で行われる（aliased）可能性のある参照の集合が作られる（図５、ブロック８２）。ブロック８６においてテストされるように、同じ配列に対してなされるかまたはその可能性がある参照は、全て１つの集合に入れられる。一度には１つのエイリアス集合しかテストされないので、このコードについてのパスが何回か実行される。このプロセスは、ステップ８２～９４のループに示されている。

【００５４】この方法では、図５のデータフロー解析ステップ９１において、２つの配列参照が同じメモリ・ロケーションを指示するものであるかどうかをテストする。この手順を図６に詳細に示す。２つの添字 $a_1J + b_1$ 及び $a_2J - b_2$ がプログラムの同じ部分にあれば、同じメモリ・ロケーションを指示するためには、$(a_1 - a_2)J + (b_1 - b_2)$ が、J がある整数値を取ったときに０にならなければならない。この方法では、２つの添字付き参照 $X(a_1J - b_1)$ と $X(a_2J - b_2)$ が同じ近隣の場所にあると確認したならば、次にこれら２つが同じメモリ・ロケーションを指すものであるかどうかをテストする。これについては、図６のステップ２０４～２２６に示し、以下に考察するように、５通りの結果が有り得る。（ここでは、一次元配列についてのみ考察するが、本発明の方法は、各次元毎に添字を別個に考えるかあるいは線形化した配列の単一の添字について考察することにより、多次元配列にも適用することが可能である。）

【００５５】１．常に違うロケーションを参照する

ステップ２０４、２０６、２１０、２１２及び２１４に

示すように、少なくとも次の３つの場合には、２つの参照は同じメモリ・ロケーションを指さない：

(1)（ステップ２０４，２０６）$(a_1 - a_2) = 0$ だが $(b_1 - b_2) \neq 0$（ここで # は不等号である。以下同様）

(2)（ステップ２０４，２１０）$(a_1 - a_2) \neq 0$ であり $(b_1 - b_2)$ は $(a_1 - a_2)$ で割り切れない

(3)（ステップ２０４，２１０，２１４）$(a_1 - a_2) \neq 0$ であり $(b_1 - b_2)$ は $(a_1 - a_2)$ で割り切れるが、$(b_1 - b_2)(a_1 - a_2)$ は J の値が取る範囲外にあると証明することができる。

【００５６】２．常に同じロケーションを参照する

$(a_1 - a_2) = 0$ かつ $(b_1 - b_2)$（ステップ２０４～２０８）ならば、２つの参照は J の値にかかわらず同じメモリ・ロケーションに対するものである。従って、それら２つの参照はテキスト上は同じである。

【００５７】３．繰り返し同じロケーションを参照する

$(a_1 - a_2) \neq 0$ で $(b_1 - b_2)$ が $(a_1 - a_2)$ で割り切れ、かつ $(b_1 - b_2)(a_1 - a_2)$ が J の値が取る範囲内にあることが証明できれば、２つの参照が同じメモリ・ロケーションに対してなされる J の値が１つある。更に、J は、ステップ２０４、２１０、２１４、２１６、２２０に示すように、この値を繰り返して取る。

【００５８】４．過渡的に同じロケーションを参照する

$(a_1 - a_2) \neq 0$ かつ $(b_1 - b_2)$ が $(a_1 - a_2)$ で割り切れかつ $(b_1 - b_2)(a_1 - a_2)$ が J の値が取る範囲内にあると証明できれば、２つの参照が同じメモリ・ロケーションに対してなされるようになる J の値が１つある。しかし、ステップ２１４及び２１８に示すように、J はその値を１度だけ取る。J がループ・インデクスであるならば、これは、予期された結果である。

【００５９】５．不確定

時には、この方法で参照が上記の４つのカテゴリのどれかに入ることを証明することができない場合がある。例えば、ステップ２００及び２０２に示すように、不確定の結果は、２つの参照がエイリアス同士であるかも知れない配列に対するものであったり、２つの参照が同じ配列に対するものであるが添字変数が異なっていたり、あるいは２つの添字の差が正規化された線形の形式になっていない場合に生じる。

【００６０】本発明の方法においては、２つのメモリ・ロケーションは、上記１．（常に違うロケーションを参照する）の場合は「異なる」と見なされる（ステップ２２４）。また、２．（常に同じロケーションを参照する）の場合は「同じである」と見なされる（ステップ２２６）。その他の場合は「同じかもしれない」と見なされる（ステップ２２２）。別々のメモリ・ロケーションに対する複数の参照は独立しており、並列実行が可能である。一貫して同じメモリ・ロケーションに対する複数の参照は常に依存関係にあり、並行性を妨げるが、ロード及びストアについての最適化の機会が得られることがしばしばある。残り３つの場合は、結果がいつも予測可能ではないので検討する価値はほとんどなく、「依存関係を有かもしれない」と分類される。

【００６１】データフロー解析

本発明による最適化は、次に図７に示すようなデータフロー解析へ移る。当技術分野においては周知のように、データフロー解析は、個々の基本コード・ブロック間のプログラムフローに対応しこのプログラムフローを表したデータフロー方程式を構築することによって行われる。ＩＬＰコンパイラでの添字付き参照の最適化を効率的にするために、本発明では、更にコンパイラ・プロセッサ・メモリにコンパイラ・マップを設ける。コンパイラ・マップは、プログラムの任意の点におけるメモリ・ロケーションと拡張仮想レジスタとの間の関係を記述する。好ましくは、このマップは <X(f(I)), y[n]> の形式のマップ・タップルの集合よりなるデータ構造にストアする。マップ・タップルの２つのエレメントには各々Ｍ－名及びＲ－名がラベルされる。マップ・タップルは、プログラムのその特定の点でテキスト表現された名前 X(f(I)) によってアドレス可能であるメモリ・ロケーションが、その同じ特定の点におけるその時の名称が t[n] である拡張仮想レジスタのエレメントと対応しているということを示す。マップ・タップルはメモリ中では整数対によって表すことができる。

【００６２】ここで、S はその中のすべてのタップルが「メモリ・ロケーション X(f(I))の内容は、現在のところ、対応するＥＶＲエレメント t[n] に入っている」というような同じ性質を有するようなマップ・タップルの集合であるとする。このような集合は従来のデータフロー・アルゴリズムによって扱うことも可能である。しかしながら、ステップ２４０でＩＲステートメントを調べた後、本発明の方法においては、図７に示す５つの特別なデータフロー解析ステップを用いる。

【００６３】１．マップ・タップルの挿入及び削除（ステップ２４２～２４３）

Ｍ－名 X(f(J)) と同じかまたは同じである可能性があるメモリ・ロケーションへのストアが行われるプログラム中の各点で、マップ・タップル (X(f(J)), t[n]) を S から削除する。ロード操作が X(f(J)) を t[n] にロードするか、あるいはストア操作が t[n] をＭ－名 X(f(J)) へストアするプログラム中の各点で、マップ・タップルX(f(J)), t[n])を集合 S に挿入する。

【００６４】２．メモリ・ロケーションの再命名

コンパイラがソース・コードステートメントをＩＲの表現式に翻訳する際、集合 S は前方に伝播される。コンパイラが J = g(J) の形式のステートメントに遭遇すると（ステップ２４４）、ステップ２４６に示すように、Ｍ－名が J の関数である全てのマップ・タップル <X(f(J)), t[n]> が削除される。関数 g(.)が許容的であり

(admissible)（ステップ２４８）、前方フロー解析が行われていると（ステップ２４９）、マップ・タップル $\langle X(f(g^{-1}(J))), t[n]\rangle$ が集合に付加される（ステップ２５０）。関数 g(.) は、それが何らかのループ不変量によるインクリメント操作またはデクリメント操作に対応する場合にのみ許容的であると定義される。例えば、X(J+5) がステートメント J = J+2 の前にアクセス可能であれば、X(J+3) はそのステートメントの後にアクセス可能であり、従って X(J) は許容可能である。同様に、集合 S が J = g(J) の形式のステートメントを越えて後方へ伝播する場合は必ず、そのＭ－名が J の関数である全てのマップ・タップル $\langle X(f(J)), t[n]\rangle$ が削除され、また関数 g(.) が許容的であれば、マップ・タップル $\langle X(f(g(J))), t[n]\rangle$ が集合に付加される。

【００６５】３．拡張仮想レジスタ・エレメントの再命名

集合 S が remap(t) の形式のステートメントを越えて前方に伝播すると、ステップ２５８に示すように、Ｒ－名が t のエレメントである全てのマップ・タップル $\langle X(F(J)), t[n]\rangle$ がマップ・タップル $\langle X(F(J)), t[n+1]\rangle$ で置換される。同様に，ステップ２５６に示すように，集合 S が remap(t) の形式のステートメントを越えて後方へ伝播する場合は必ず、Ｒ－名が t のエレメントである全てのマップ・タップル $\langle X(f(J)), t[n]\rangle$ が $\langle X(f(J)), t[n-1]\rangle$ で置換される。

【００６６】４．合流操作（meet operation）（ステップ２６０）

プログラム中で n 本の制御フローパスが出会う点において、これらのパスに沿って伝播する集合を S_i（i = 1, ..., n）で表す。これらのパスの合流点をちょうど越えた点での集合 S は次のようにして構成される。合流操作が論理積型の場合は、全ての入来パス i 上の集合 S_i 中に $\langle X(f(J)), ti\rangle$ の形式のマップ・タップルがある場合にのみ、Ｍ－名 X(f(J)) を有するマップ・タップルが Sに入れられる。言い換えると、Ｍ－名が論理積型合流を通って伝播するためには、同じＭ－名が各パスから到達しなければならない。更に、これが成立するとき、マッチングする全ての入来マップ・タップルの ti がＥＶＲ名称においてもインデクスにおいても同じである場合は、同じＲ－名が出力マップ・タップルに入れられる。そうでない場合は、新しいＥＶＲ r がメモリ中で割り当てられ、r[0] が出力マップ・タップルのＭ－名として用いられる。合流操作が論理和型の場合は、いずれかの入来パスから入ってきた全ての入来マップ・タップルが出力集合 S に入れられる。

【００６７】これらの操作を用いて、プログラムの１つの点からのマップ・タップルは、プログラム中の同じ点で２つの配列添字の比較を行うことができるように、プログラム中の他の点に正しく伝播される。

【００６８】その後、この方法では、問題とされている

全ての点において集合 S_i の値を、安定した解が見付かるまで繰返し計算することにより、従来のデータフロー解析を実行する。プログラム中の任意の点での集合 S は、どのＭ－名がＥＶＲでアクセス可能であり、どの特定のＥＶＲエレメント中にそれらのメモリ・ロケーションの内容が見出されるかを正確に指示する。

【００６９】ここで、フロー方程式を解く反復プロセスが安定状態に収束するかどうかという疑問を抱くかも知れない。I = g(I) の形式のステートメントに遭遇する都度行われるＭ－名の再命名があるとすると、集合 S は、ループの逐次反復の間に伝播するに従い際限なく成長であろう。５つのデータフロー解析ステップの後、ステップ２６２でウィンドウ化（windowing）技法を適用することによりメモリ・オーバフローを防ぐことができる。

【００７０】例えば添字変数が単調に変化する場合、すなわち I がインクリメントするだけか、あるいはデクリメントするだけの場合は、集合 S の際限のない成長は添字に対して最大値及び最小値を限定することにより抑止することができる。再命名の後、Ｍ－名がこれらの値のウィンドウの外に出た際には、そのマップ・タップルはマップ・タップルの集合から必ず放棄される。コンパイラが解析中のコードの領域内の全てのテキスト参照を含むようにウィンドウを設定すれば、有用な情報は失われない。Ｍ－名がいったんウィンドウの外に出て行ってしまうと、それ以後のループを回っての伝播はそのＭ－名をウィンドウの更に外側へ移動させるだけであり、プログラム中のどの参照とも同じになることは二度とない。従って、このＭ－名は値を持たず、放棄することができる。このように、ウィンドウを設定すると、マップ・タップル集合の「切捨て」が可能となり、反復プロセスを解へ向けて収束させることができる。

【００７１】上述のようにウィンドウを設定することができない場合は、その代わりに、マップ・タップル集合を一定回数Ｎだけループ回を回す。これによって、最大Ｎ回だけ離れた反復同士の間の依存関係に関する正確な情報を含むマップ・タップル集合が得られる。この回数を越えた先については情報は全く得られず、最悪の場合は、ある反復における全ての参照が反復回数Ｎ回以上離れた全ての参照に依存すると考えることも可能である。実際の使用に関しては、本願中で開示した方法は最適化されていないコードにも適用することができ、その場合各操作毎にメモリからソース・オペランドをロードし、結果を逆にメモリに保管して、メモリ・ロードが最適化されたＤＳＡ形式へのコードの変換を行わせる。

【００７２】これを行うためには、上に説明したデータフロー解析の間に、S を既知のＥＶＲ中でアクセス可能であるＭ－名をリストするマップ・タップルの集合と定義する。次に、冗長なロードをＩＲロード中のテキスト表現された名前に既知のＥＶＲエレメントを代入するこ

23

とによって除去する。その近隣についてのマップ・タップル集合中に対応するマップ・タップル〈V,v〉を有するM－名 V を指定するロード操作を除去することができる。それは、そのメモリ・ロケーションの内容は既知のＥＶＲエレメントにあるからである。そのロード操作の結果を使用することになっている操作は、ＥＶＲエレメント v を参照するよう変更される。

【００７３】冗長なストア操作を除去するためには、等価な後退フロー解析を用いることができる。例えば、プログラム中で V に「等しいかもしれない」変数に対するストアをおこなう点において、M－名として V を有する全てのマップ・タップルは、プログラム中のその点における集合 S から削除される。

【００７４】データフロー解析は、上述のように、マップ・タップルを集合 S に対して加除しつつ続けられる。全てのステートメントの処理が終わると、プログラム中の任意の点における集合 S によって、どのM－名がＥＶＲ中でアクセス可能であるか、そしてそれらのメモリ・ロケーションの内容がどの特定のＥＶＲエレメント中にあるかが正確に指示される。

【００７５】これ以外の本発明の説明は、「命令レベル並行処理のためのデータフロー及び依存性解析（Data Flow and Dependence Analysis for Instruction-Level Parallelism)」という名称の論文に記載されている。また、本発明は、本願に特定的に開示した以外の形で実施することができる。従って、本発明の範囲は特許請求の範囲の記載のみによって限定されるものである。

【００７６】

24

【効果】以上詳細に説明したように、本発明によれば、命令レベル・パラレル・プロセッサのロードの最適化を行うことができる。

【図面の簡単な説明】

【図１】コンパイラとプロセッサのブロック図。

【図２】拡張仮想レジスタとremap関数を使用する前と後で仮想拡張コンパイラにストアされている値を説明する図。

【図３】最適化方法のフローチャート。

【図４】図３のフローチャートの部分ステップとなる動的単一代入変換方法のフローチャート。

【図５】図３のフローチャートの部分ステップとなる依存性解析と最適化のフローチャート。

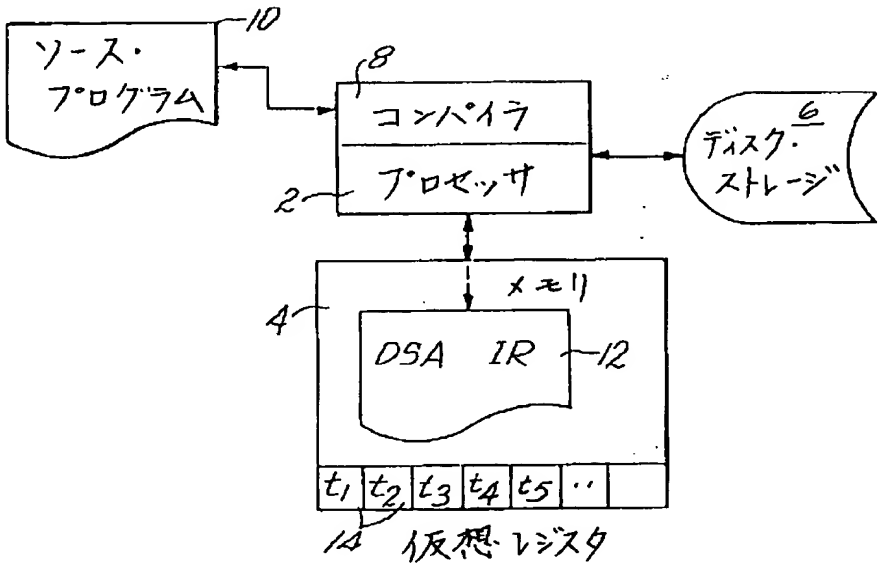
【図６】図３のフローチャートの部分ステップとなるメモリロケーション一貫性テストのフローチャート。

【図７】図３のフローチャートの部分ステップとなるデータフロー解析方法のフローチャート。

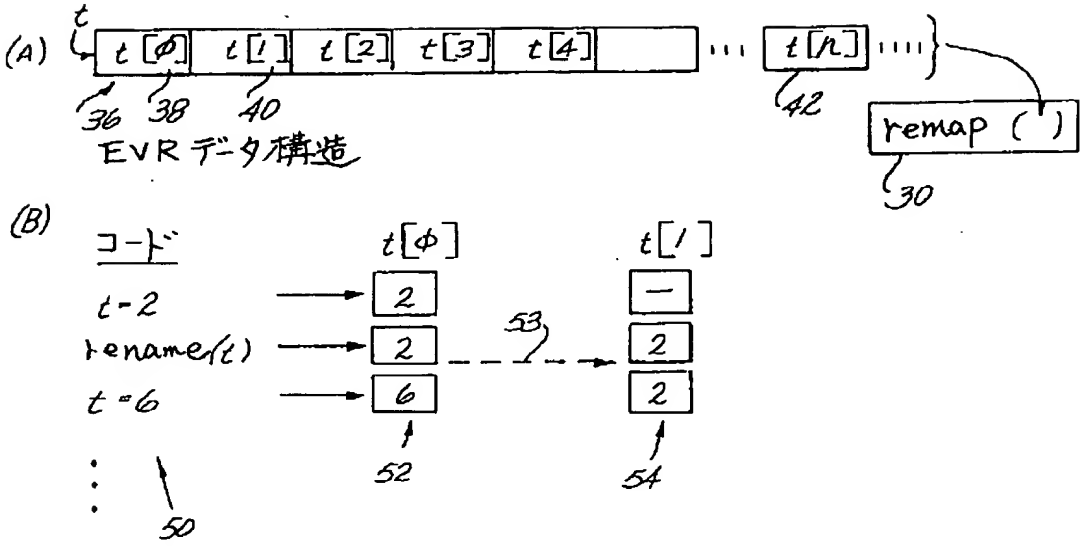
【符号の説明】

２：プロセッサ
４：メモリ
６：ディスク・ストレージ
８：コンパイラ
１０：ソース・プログラム
１２：ＤＳＡ ＩＲ
１４：仮想レジスタ
３６：ＥＶＲデータ構造
３８、４０、４２：仮想レジスタ・エレメント

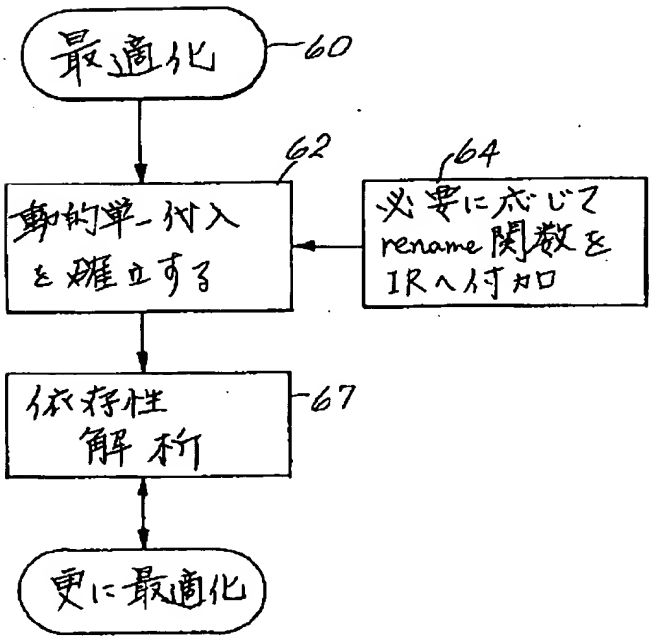
【図１】



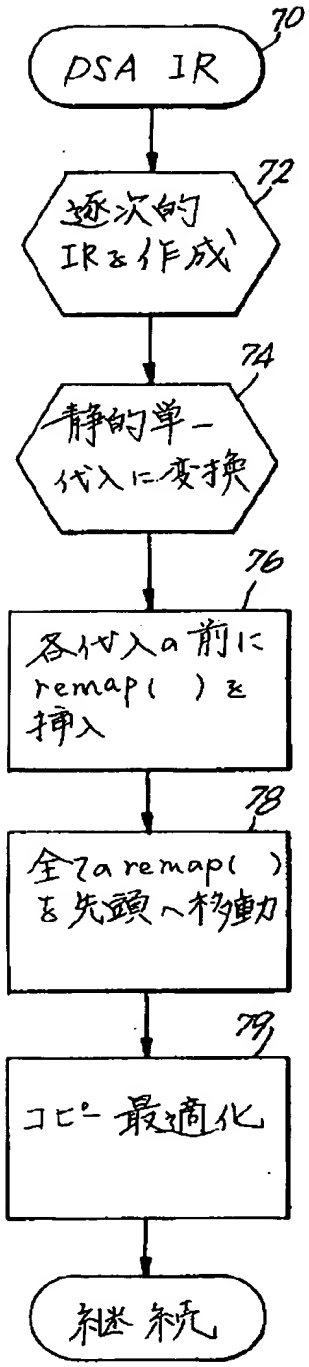
【図2】


(A)
t
t[0]
t[1]
t[2]
t[3]
t[4]
t[n]
36
38
40
42
EVRデータ構造
remap ( )
30
(B)
コード
t[0]
t[1]
t=2
rename(t)
t=6
2
2
6
—
2
2
53
52
54
50

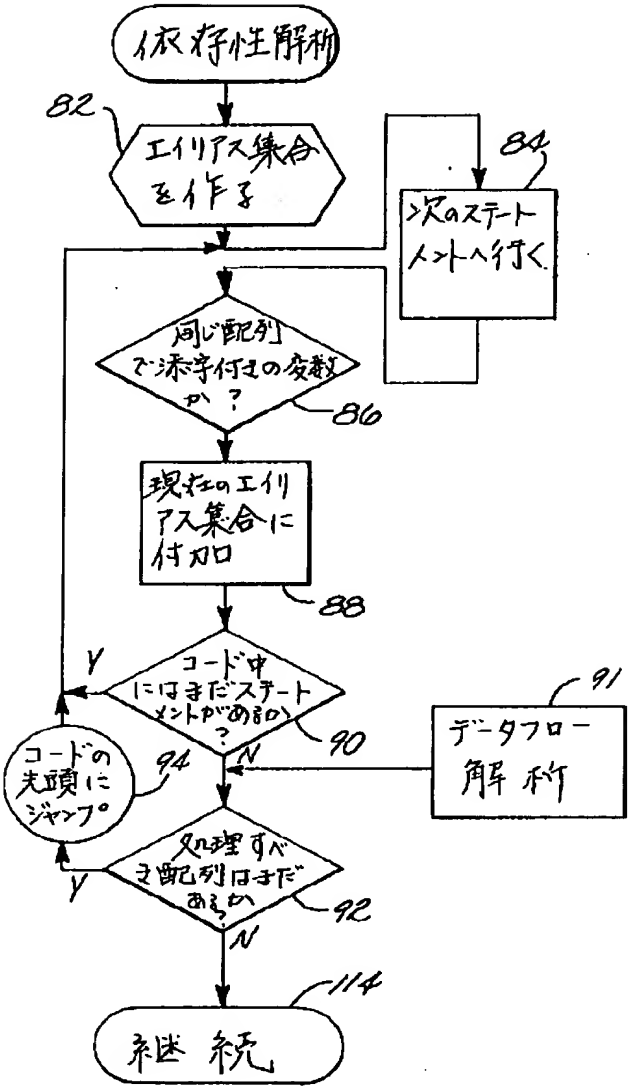
【図3】


最適化
60
62
64
動的単一代入を確立する
必要に応じてrename関数をIRへ付加
依存性解析
67
更に最適化

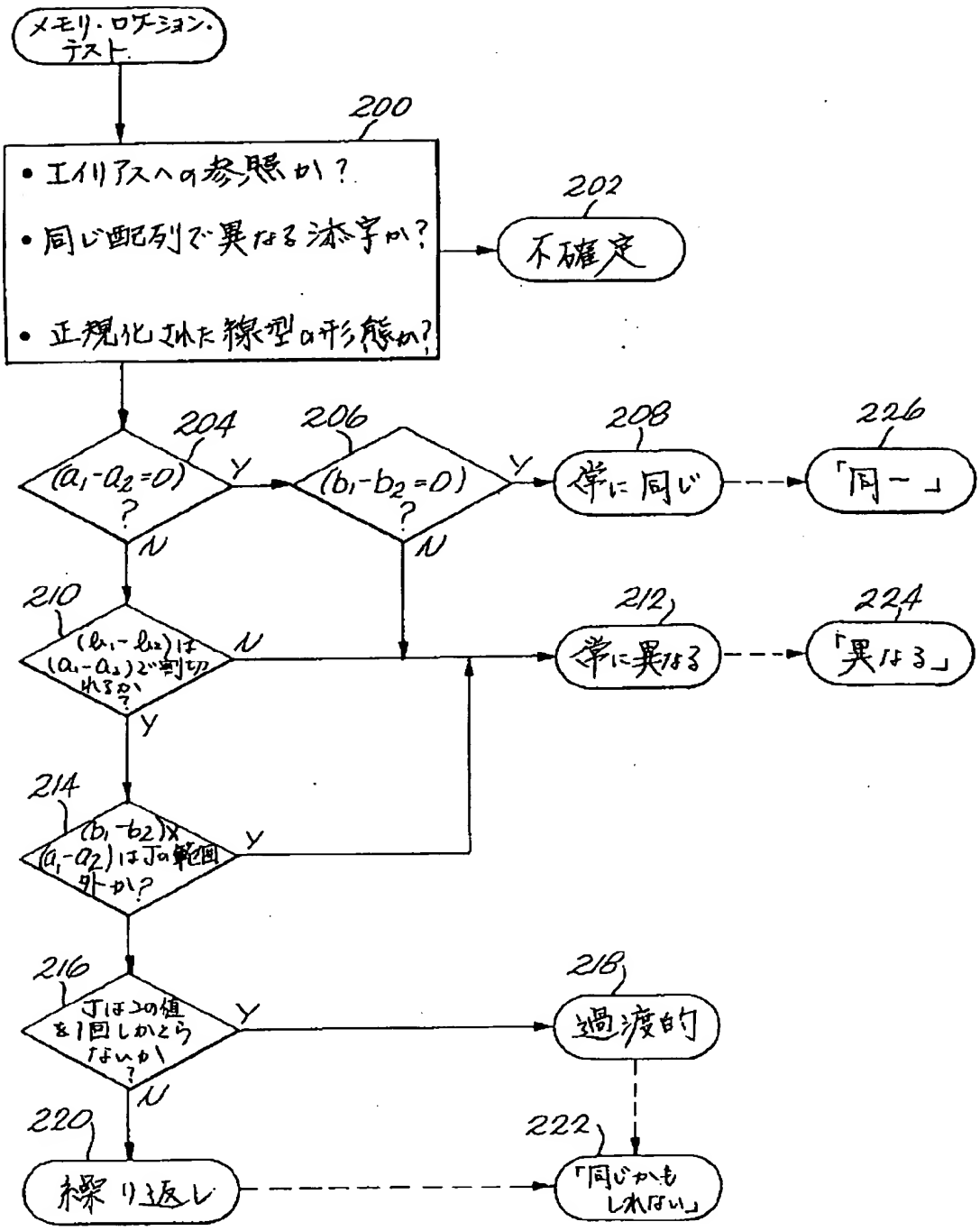
【図４】


DSA IR
70
逐次的IRを作成
72
静的単一代入に変換
74
各代入の前にremap( )を挿入
76
全てのremap( )を先頭へ移動
78
コピー最適化
79
継続

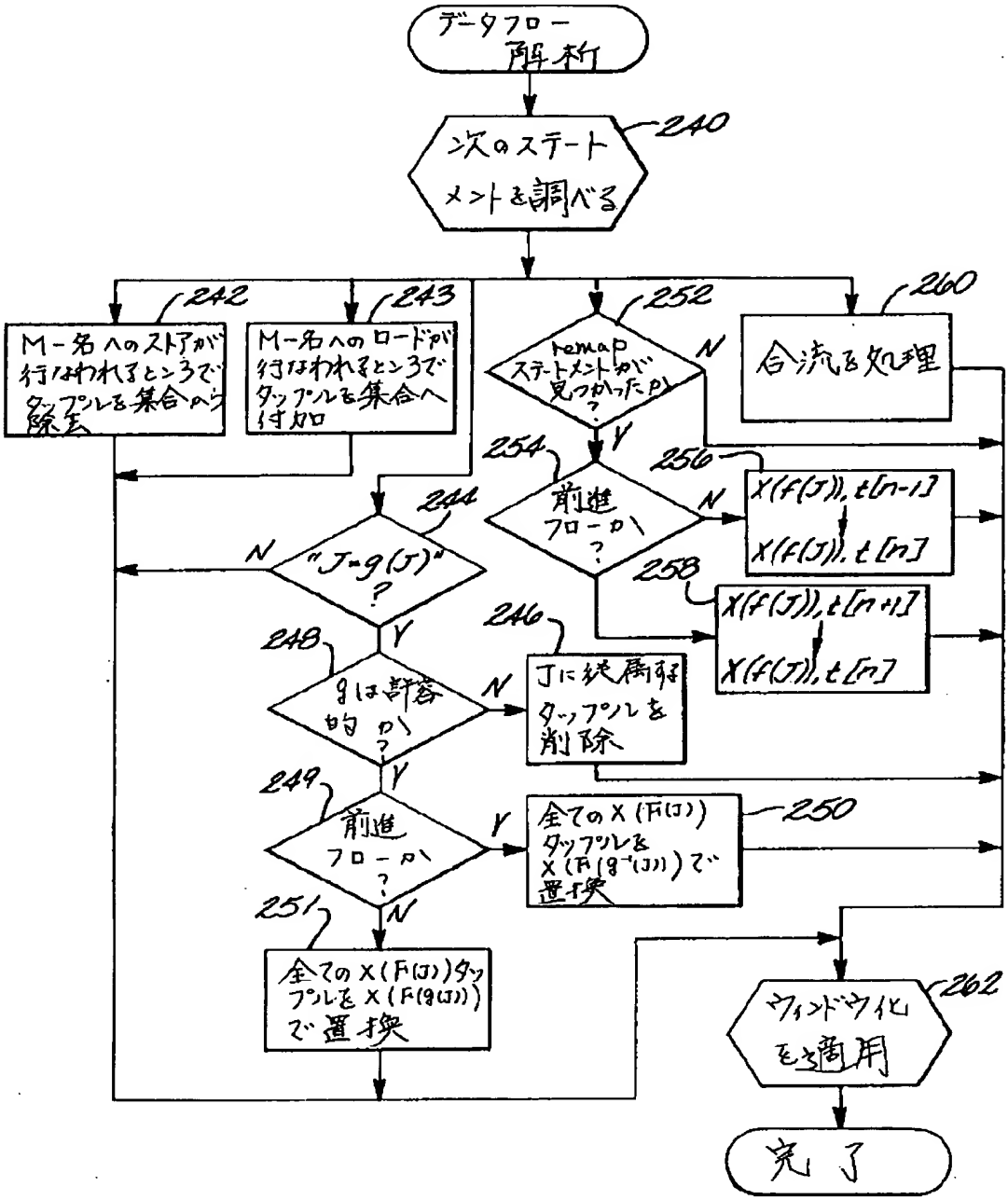
【図５】


依存性解析
82
エイリアス集合を作る
84
次のステートメントへ行く
同じ配列で添字付きの変数か？
86
現在のエイリアス集合に付加
88
コード中にはまだステートメントがあるか？
90
Y
N
データフロー解析
91
コードの先頭にジャンプ
94
処理すべき配列はまだあるか
92
Y
N
継続
114

【図6】


メモリ・ロケーション・テスト
200
• エイリアスへの参照か？
• 同じ配列で異なる添字か？
• 正規化された線型の形態か？
202
不確定
204
(a1-a2=0)？
Y
N
206
(b1-b2=0)？
Y
N
208
常に同じ
226
「同一」
210
(b1-b2)は(a1-a2)で割切れるか？
N
Y
212
常に異なる
224
「異なる」
214
(b1-b2)×(a1-a2)は J の範囲外か？
Y
216
J はこの値を1回しかとらないか？
Y
N
218
過渡的
220
繰り返し
222
「同じかもしれない」

【図７】


データフロー解析
次のステートメントを調べる
240
242
M－名へのストアが行なわれるところでタップルを集合から除去
243
M－名へのロードが行なわれるところでタップルを集合へ付加
252
remap ステートメントが見つかったか？
N
Y
260
合流を処理
254
前進フローか？
256
X(F(J)), t[n-1]
X(F(J)), t[n]
244
"J=g(J)"？
N
Y
258
X(F(J)), t[n+1]
X(F(J)), t[n]
246
J に依存するタップルを削除
248
g は許容的か？
N
Y
249
前進フローか？
Y
全ての X(F(J)) タップルを X(F(g⁻¹(J))) で置換
250
251
全ての X(F(J)) タップルを X(F(g(J))) で置換
ウィンドウ化を適用
262
完了